

















# **Numere mari. Aplicații**

*Alexandru Cohal  
Noiembrie 2013*

## Cuprins

 <i>Introducere .....</i>	3
 <i>Reprezentarea numerelor mari .....</i>	4
 <i>Citirea unui număr mare .....</i>	5
 <i>Afișarea unui număr mare .....</i>	6
 <i>Compararea a două numere mari.....</i>	7
 <i>Suma a două numere mari .....</i>	8
 <i>Diferența a două numere mari.....</i>	9
 <i>Produsul dintre un număr mare și o putere a lui 10 .....</i>	10
 <i>Produsul dintre un număr mare și un număr mic.....</i>	11
 <i>Produsul a două numere mari .....</i>	12
 <i>Împărțirea dintre un număr mare și o putere a lui 10.....</i>	13
 <i>Împărțirea dintre un număr mare și un număr mic .....</i>	14
 <i>Împărțirea a două numere mari.....</i>	15
 <i>Probleme.....</i>	16
 <i>Legături .....</i>	16
 <i>Bibliografie.....</i>	16



## Introducere

Dacă într-o problemă avem nevoie să lucrăm cu numere naturale mai mari decât  $2^{64}$  nu putem folosi niciunul dintre tipurile de date predefinite ale limbajului C/C++.

Tipurile de date cele mai mari ale limbajului C/C++ sunt cele pe 8 *bytes* (octeți):

- **long long** care poate reține valori în intervalul  
[ -9.223.372.036.854.775.808 , 9.223.372.036.854.775.807 ] =  $[-2^{63}, 2^{63} - 1]$
- **unsigned long long** care poate reține valori în intervalul  
[ 0 , 18.446.744.073.709.551.615 ] =  $[0, 2^{64} - 1]$ .

Prin urmare, trebuie să implementăm propriul nostru tip de date pentru *numere mari* precum și funcțiile care să realizeze operațiile (comparare, adunare, scădere, înmulțire, împărțire) cu astfel de *numere mari*.



## Reprezentarea numerelor mari

Vom reprezenta un *număr natural mare* ca pe un vector în care reținem în ordine cifrele sale începând cu unitățile.

### Exemplu:

Numărul 1234 va fi reprezentat astfel:

0	1	2	3
4	3	2	1

### Observații:

➡ Se observă faptul că indicele poziției din vector al oricărei cifre coincide cu puterea bazei corespunzătoare acelei cifre.

Să considerăm că *numerele mari* cu care vom lucra în continuare au maxim 100 de cifre. Declararea unui astfel de *număr mare* este:

```
char NrMare [DIMMAX] ;
```

### Observații:

➡ **DIMMAX** este o constantă căreia i-a fost atribuită valoarea **100** reprezentând numărul maxim de cifre al unui număr mare.

➡ Tipul elementelor vectorului este **char** deoarece în fiecare poziție a vectorului va fi memorată o cifră (dacă am fi folosit tipul **int** sau alt tip care să necesite mai mult de 1 **byte** am fi irosit memoria).



## Citirea unui număr mare

Pentru a citi un *număr mare* vom citi de la tastatură întreg numărul într-un șir de caractere, apoi vom transforma caracterele cifră în numere și le vom memora în ordine inversă în vectorul ce va conține cifrele *numărului mare*.

Pentru a ușura calculele viitoare vom completa elemente vectorului care nu au fost ocupate de cifrele numărului mare cu valoarea **0**. (Dacă vectorul este declarat global atunci nu mai este nevoie de acest pas întrucât toate elementele sale sunt inițializate automat cu valoarea **0**).

Funcția **citire()** va avea doi parametri:

- Vectorul în care reținem cifrele *numărului mare* citit: **char a[]**
- Numărul de cifre ale *numărului mare*: **int &lga**

### Observații:

➡ Am transmis parametrul **lga** prin referință pentru ca funcția **citire()** să poată modifica valoarea acestui parametru, valoarea rămânând modificată și după apel.

➡ Parametrul **a** nu este necesar să fie transmis prin referință, deoarece a este numele unui vector, fiind un pointer constant la primul element al vectorului.

➡ Este necesar ca funcția **citire()** să aibă acești doi parametri pentru cazul în care vrem să citim mai multe *numere mari*, apelând astfel funcția pentru fiecare *număr mare* în parte.

```
void citire(char a[], int &lga)
{
    char s[DIMMAX + 1];
    int i;

    cin >> s;
    //determinam numarul de cifre
    lga = strlen(s);

    //transformam si retinem
    for (i=lga-1; i>=0; i--)
        a[lga - i - 1] = s[i] - '0';

    //completam cu 0
    for (i=lga; i<DIMMAX; ++i)
        a[i] = 0;
}
```



## Afișarea unui număr mare

*Numărul mare* va fi afișat astfel: Vom parcurge vectorul de cifre de la sfârșit către început, afișând succesiv fiecare cifră.

La fel ca și funcția **citire()**, funcția **afisare()** va avea doi parametri:

- Vectorul în care au fost reținute cifrele *numărului mare*: **char a[]**

- Numărul de cifre ale *numărului mare*: **int lga**, care de această dată nu mai este o referință deoarece nu va mai fi modificat.

```
void afisare(char a[], int lga)
{
    int i;

    //parcurgem si afisam
    for (i=lga-1; i>=0; --i)
        cout << (int) a[i];

    cout << '\n';
}
```

### Observații:

➡ Deoarece elementele vectorului sunt de tipul **char**, dacă am afișa cifrele numărului mare astfel:

```
cout << a[i];
```

atunci cifrele ar fi fost considerate coduri ASCII și ar fi fost afișate caracterele corespunzătoare. De aceea trebuie să facem conversia explicită de tip

```
(int) a[i]
```

și să forțăm să se afișeze cifra memorată.



## Compararea a două numere mari

Să considerăm două *numere mari* memorate în vectorii **a** și **b**. Pentru a le compara, întâi trebuie să comparăm numărul lor de cifre.

Dacă numărul de cifre al lui **a** este mai mic decât numărul de cifre al lui **b**, atunci **a** este mai mic decât **b** (**a** < **b**). Și invers: dacă numărul de cifre al lui **b** este mai mic decât numărul de cifre al lui **a**, atunci **b** este mai mic decât **a** (**a** > **b**).

Dacă numerele **a** și **b** au același număr de cifre, atunci parcurgem cele două numere începând de la cifra cea mai semnificativă (cea cu indicele cel mai mare din vector) până la întâlnirea a două cifre distincte. Dacă am întâlnit două cifre diferite, ele determină ordinea dintre numerele **a** și **b**. În caz contrar, numerele sunt egale.

Funcția **comparare()** va avea 4 parametri:

- Cei doi vectori în care au fost memorate cifrele celor două *numere mari*
- Numărul de cifre ale celor două numere.

Funcția **comparare()** va returna o valoare care poate fi:

- **-1** dacă primul număr dat ca parametru este mai mare decât celalalt (**a** < **b**)
- **1** dacă al doilea număr dat ca parametru este mai mare decât celalalt (**a** > **b**)
- **0** dacă cele două numere sunt egale (**a** = **b**).

```
int comparare(char a[], int lga, char b[], int lgb)
{
    int i;

    //daca a are mai putine cifre decat b
    if (lga < lgb)
        return -1;

    //daca b are mai putine cifre decat a
    if (lgb < lga)
        return 1;

    //daca au acelasi numar de cifre, incep sa le compar
    for (i=lga-1; i>=0; --i)
        if (a[i] < b[i])
            return -1;
        else
            if (a[i] > b[i])
                return 1;

    //daca nu am gasit nicio cifra diferita
    return 0;
}
```



## Suma a două numere mari

Pentru a calcula suma a două *numere mari* vom parcurge simultan cele două numere, începând de la unități (indicele **0** al vectorilor). Vom aduna cele două cifre de pe poziții corespondente și vom lua în calcul și eventuala *cifra de transport* care s-a obținut de la adunarea precedentă. Reținem în sumă cifra obținută prin adunare (restul împărțirii rezultatului adunării celor două cifre și a transportului la 10) și recalculăm transportul (câtuș împărțirii rezultatului adunării celor două cifre și a transportului la 10).

Funcția **adunare()** va avea 6 parametri:

- Cei doi vectori în care au fost memorate cifrele celor două numere pe care vrem să le adunăm și numărul lor de cifre
- Vectorul în care reținem rezultatul adunării (**char suma[]**)
- Numărul de cifre al sumei (**int &lgsoma**) care va fi transmis prin referință.

```
void adunare(char a[], int lga, char b[], int lgb, char suma[],
             int &lgsoma)
{
    int i, t = 0;

    //initializam lungimea vectorului suma cu maximul dintre
    //lungimile celor doua numere
    lgsoma = max(lga, lgb);

    //parcurgem numerele cifra cu cifra si adunam
    for (i=0; i<lgsoma; ++i)
    {
        suma[i] = t + a[i] + b[i];
        t = suma[i] / 10;
        suma[i] = suma[i] % 10;
    }

    //daca la sfarsit obtinem un transport nenul, mai avem o cifra
    if (t)
        suma[lgsoma++] = t;
}
```

### Observații:

➡ Suma ar putea avea o cifră în plus față de cel mai mare dintre numere (dacă la sfârșitul adunării cifra de transport este nenulă). Din această cauză, trebuie să declarăm vectorul **suma** cu un element în plus față de maximul dintre numărul de elemente ale numerelor pe care vrem să le adunăm (dacă vectorii **a** și **b** au fost declarați ca având **DIMMAX** elemente, atunci vectorul **suma** trebuie să aibă **DIMMAX + 1** elemente).

➡ Pentru că la citirea unui *număr mare* am completat elementele vectorului care nu au fost ocupate de cifrele *numărului mare* cu valoarea **0** putem efectua fără grija alterării rezultatului adunarea cifră cu cifră chiar dacă cele două numere au lungimi diferite.

### Exemplu:

Suma dintre numerele  $a = 1234$  și  $b = 9999$  va arăta astfel:

	0	1	2	3	4
<i>a</i>	4	3	2	1	0
<i>b</i>	9	9	9	9	0
<i>a+b</i>	3	3	2	1	1





## Diferența a două numere mari

Vom presupune că *descăzutul* (numărul memorat în vectorul **a**) este mai mare decât *scăzătorul* (numărul memorat în vectorul **b**).

Pentru a calcula diferența a două numere mari vom parcurge descăzutul începând de la unități (indicele **0** al vectorului). Vom scădea din cifra curentă a descăzutului cifra curentă a scăzătorului și vom lua în calcul și eventuala cifră de transport obținută de la scăderea precedentă.

- Dacă rezultatul este negativ, “împrumutăm” **10** de pe poziția următoare (este posibil întrucât descăzutul este mai mare decât scăzătorul) și în acest caz cifra de transport devine **-1**

- Dacă rezultatul este pozitiv, nu avem nevoie de “împrumut” și cifra de transport devine **0**.

Diferența ar putea avea mai puține cifre decât descăzutul, astfel încât la sfârșit verificăm dacă avem zerouri nesemnificative la începutul numărului și actualizăm numărul de cifre al rezultatului astfel încât prima cifră a vectorului **diferenta** (cea cu indicele cel mai mare) să fie diferită de zero.

Parametrii funcției **scadere()** sunt similari cu cei ai funcției de adunare (**adunare()**).

```
void scadere(char a[], int lga, char b[], int lgb, char diferenta[],
             int &lgdiferenta)
{
    int i, t = 0;

    //initializam lungimea vectorului diferenta
    lgdiferenta = lga;

    //parcurgem numerele cifra cu cifra si scadem
    for (i=0; i<lgdiferenta; ++i)
    {
        diferenta[i] = a[i] - b[i] + t;
        if (diferenta[i] < 0)
        {
            diferenta[i] += 10;
            t = -1;
        }
        else
            t = 0;
    }

    //verificam daca avem zerouri nesemnificative la inceputul
    rezultatului
    i = lgdiferenta - 1;
    while (diferenta[i] == 0)
        i--;

    //actualizam lungimea vectorului diferenta
    lgdiferenta = i + 1;
}
```



## Produsul dintre un număr mare și o putere a lui 10

Pentru a înmulți un *număr mare* cu  $10^{nr}$  trebuie să adăugăm la sfârșitul *numărului mare* **nr** zerouri, adică să deplasăm cifrele *numărului mare* reținute în vectorul **a** cu **nr** poziții la dreapta și să completăm pozițiile eliberate cu zerouri.

Funcția **inmultirePutere10()** va avea 5 parametri:

- Vectorul în care a fost memorat *numărul mare* (**char a[]**) și numărul lui de cifre (**int lga**)
- Puterea lui 10 cu care înmulțim *numărul mare* (**int nr**)
- Vectorul în care depunem calculul (**char sol[]**) și numărul lui de cifre (**int &lgsol**) (trimis prin referință).

```
void inmultirePutere10(char a[], int lga, int nr, char sol[], int &lgsol)
{
    int i;

    //completez cu zerouri
    for (i=0; i<nr; ++i)
        sol[i] = 0;

    //copii cifrele numarului mare
    for (i=0; i<lga; ++i)
        sol[nr + i] = a[i];

    //numarul de cifre al rezultatului
    lgsol = nr + lga;
}
```

### Exemplu:

Produsul dintre numărul  $a = 1234$  și  $10^3$  va arăta astfel:

	0	1	2	3	4	5	6
$a$	4	3	2	1	0	0	0
$a \cdot 10^3$	0	0	0	4	3	2	1



## ***Produsul dintre un număr mare și un număr mic***

Pentru a înmulți un *număr mare* cu un număr mic (care se încadrează într-un tip de date al limbajului C/C++) putem aplica o metoda mai ușoară decât să trecem numărul mic la reprezentarea *numerelor mari* și să aplicăm produsul dintre două *numere mari*.

Se înmulțește pe rând, începând cu cifra unităților, fiecare cifră a *numărului mare* cu numărul mic, adunând un eventual transport de la înmulțirea precedentă.

La sfârșit, dacă mai avem un transport care nu a fost cuprins în rezultat, îl copiem cifră cu cifră în continuarea rezultatului.

Parametrii funcției **inmultireMic()** sunt similari cu cei ai înmulțirii unui număr mare cu o putere a lui 10 (**inmultirePutere10()**).

```
void inmultireMic(char a[], int lga, int nr, char produsMic[],
                 int &lgprodusMic)
{
    int i, cifra, t = 0;

    //parcurgem si inmultim
    for (i=0; i<lga; ++i)
    {
        cifra = a[i] * nr + t;
        t = cifra / 10;
        produsMic[i] = cifra % 10;
    }

    //initializam lungimea rezultatului
    lgprodusMic = lga;

    //daca mai avem transport, il adaugam la rezultat
    while (t != 0)
    {
        produsMic[lgprodusMic++] = t % 10;
        t = t / 10;
    }
}
```

### ***Observații:***

➡ Și la acest tip de înmulțire (ca și la înmulțirea a două *numere mari*) trebuie să avem grijă la declararea dimensiunii vectorului rezultat (care va fi aleasă în funcție de mărimea numărului mic).



## Produsul a două numere mari

La început, vom inițializa vectorul de cifre ale produsului cu 0.

Pentru a calcula produsul dintre două *numere mari* vom înmulți fiecare cifră a deînmulțitului (numărul memorat în vectorul **a**), pe rând, cu fiecare cifră a înmulțitorului (numărul memorat în vectorul **b**) și vom aduna fiecare produs parțial obținut la rezultatul final (ale cărui cifre sunt memorate în vectorul produs), pe poziția corespunzătoare.

Atunci când înmulțim cifra **a[j]** cu toate cifrele numărului **b**, produsul parțial obținut este înmulțit cu  $10^j$ , adică cifrele sunt deplasate cu **j** poziții la dreapta.

Parametrii funcției **inmultire()** sunt similari cu cei ai funcției de adunare (**adunare()**) și de scădere (**scadere()**).

```
void inmultire(char a[], int lga, char b[], int lgb, char produs[],
               int &lgprodus)
{
    int i, j, t;

    //initializam cifrele vectorului cu 0
    for (i=0; i<DIMMAX; ++i)
        produs[i] = 0;

    //parcurgem cele doua numere si calculam
    for (i=0; i<lga; ++i)
    {
        t = 0;

        for (j=0; j<lgb; ++j)
        {
            produs[i + j] = produs[i + j] + a[i] * b[j] + t;
            t = produs[i + j] / 10;
            produs[i + j] = produs[i + j] % 10;
        }

        //daca ne ramane un transport la finalul unui produs
        partial, il punem pe pozitia urmatoare
        if (t)
            produs[i + j] = t;
    }

    //stabilim lungimea rezultatului
    lgprodus = lga + lgb - 1;

    if (produs[lgprodus])
        lgprodus++;
}
```

### Observații:

► Lungimea maximă a produsului este egala cu suma lungimilor celor doi termeni pe care îi înmulțim. Dacă cei doi termeni ocupă în totalitate toate pozițiile vectorului (adică au **DIMMAX** cifre) atunci produsul nostru poate avea **DIMMAX \* 2** cifre. Așadar, și la înmulțire (ca și la adunare) trebuie să avem grijă la declararea dimensiunii vectorului rezultat.



## Împărțirea dintre un număr mare și o putere a lui 10

La împărțirea unui *număr mare* la  $10^{nr}$  restul va fi compus din ultimele **nr** cifre ale *numărului mare* (cele de pe pozițiile de la 0 la **nr-1** din vectorul **a**), iar câtul va fi compus din restul cifrelor (cele de pe pozițiile de la **nr** la **lga-1** din vectorul **a**).

Dacă **nr** este mai mare decât lungimea *numărului mare*, atunci restul va fi egal cu *numărul mare* în întregime, iar câtul va fi egal cu zero.

### Observații:

► Deoarece împărțitorul este un număr mic, iar restul este întotdeauna mai mic decât împărțitorul, restul va fi și el un număr mic.

Parametrii funcției **inmultireMic()** sunt similari cu cei ai înmulțirii unui *număr mare* cu o putere a lui 10 ( **inmultirePutere10()** ), iar în plus avem variabila care reține restul (**int &rest**) (transmisă prin referință).

```
void impartirePutere10(char a[], int lga, int nr, char cat[],
                      int &lghcat, int &rest)
{
    int i;

    //determin restul
    for (i=min(nr-1, lga-1); i>=0; --i)
        rest = rest * 10 + a[i];

    //determin catul
    for (i=nr; i<lga; ++i)
        cat[i - nr] = a[i];

    //numarul de cifre ale catului
    lghcat = lga - nr;
    if (lghcat < 0)
    {
        lghcat = 1;
        cat[0] = 0;
    }
}
```

### Exemplu:

Rezultatul împărțirii numărului  $a = 1234$  la  $10^3$  va arăta astfel:

	0	1	2	3
<i>a</i>	4	3	2	1
<i>cât</i>	1	0	0	0
<i>rest</i>	234			



## Împărțirea dintre un număr mare și un număr mic

La fel ca la înmulțirea dintre un *număr mare* și un număr mic, pentru a împărți un *număr mare* la un număr mic putem aplica o metoda mai ușoară decât să trecem numărul mic la reprezentarea *numerelor mari* și să aplicăm împărțirea dintre două *numere mari*.

Aplicăm aceeași simulare de împărțire învățată la matematică, ținând cont și de faptul că restul este și el un număr mic deoarece împărțitorul este un număr mic.

Parametrii funcției **impartireMic()** sunt aceeași ca la funcția **inmultireMic()** doar că mai avem în plus o variabilă în care reținem restul (**rest**) (trimisă prin referință).

```
void impartireMic(char a[], int lga, int b, char cat[],
                 int &lgcat, int &rest)
{
    int i;

    //initializez restul si lungimea catului
    rest = 0;
    lgcat = lga;

    //simulez impartirea
    for (i=lga-1; i>=0; --i)
    {
        rest = rest * 10 + a[i];
        cat[i] = 0;

        while (b <= rest)
        {
            rest = rest - b;
            cat[i]++;
        }
    }

    //determin numarul de cifre ale catului
    while (!cat[lgcat - 1] && lgcat > 1)
        lgcat--;
}
```



## Împărțirea a două numere mari

Dacă cele două *numere mari* sunt comparabile ca dimensiune, putem simula împărțirea prin scăderi repetate. În caz contrar, această metodă este inefficientă.

Vom simula algoritmul de împărțire învățat la matematică.

Funcția **impartire()** va avea 8 parametri (vectorii corespunzători celor două *numere mari* pe care vrem să le împărțim (**a**, **b**), câtul (**cat**) și restul (**rest**) precum și numărul de cifre pentru fiecare dintre acești vectori (**lga**, **lgb**, **lgcat**, **lgrest**)). Numerele de cifre ale câtului și ale restului vor fi transmise prin referință.

```
void impartire(char a[], int lga, char b[], int lgb, char cat[],
               int &lgcat, char rest[], int &lgrest)
{
    int i;
    char aux[DIMMAX];
    int lgaux;

    //initializez lungimile
    lgrest = 0;
    lgcat = lga;

    //simulez impartirea
    for (i=lga-1; i>=0; --i)
    {
        inmultirePutere10(rest, lgrest, 1, aux, lgaux);
        copie(aux, lgaux, rest, lgrest);
        rest[0] = a[i];
        cat[i] = 0;

        //daca obtin un rest mai mare decat impartitor incep
        sa scad din rest impartitorul de cate ori pot
        while (comparare(b, lgb, rest, lgrest) != 1)
        {
            cat[i]++;
            scadere(rest, lgrest, b, lgb, rest, lgrest);
        }
    }

    //determin numarul de cifre ale catului si ale restului
    while (!cat[lgcat - 1] && lgcat > 1)
        lgcat--;
    while (!rest[lgrest - 1] && lgrest > 1)
        lgrest--;
}
```

### Observații:

➡ S-a folosit funcția **copie(char a[], int lga, char b[], int &lgb)** care copie element cu element vectorul **a** în vectorul **b** și actualizează numărul de cifre ale vectorului **b**.



## **Probleme**

- [Set](#) (Campion)
- [Sqr](#) (Campion)
- [Numar3](#) (Campion)
- [Banda10](#) (Campion)
- [Dale](#) (Campion)
- [Patrate2](#) (Infoarena)
  
- [Pomi](#) (Campion)
- [Muguri](#) (Campion)
- [Test](#) (Campion)
- [Cutii](#) (Campion)
  
- [Aliniere](#) (Campion)
- [Aladdin2](#) (Infoarena)
- [Biti2](#) (Infoarena)
- [Petrecere](#) (Campion)
- [Fib](#) (Campion)
- [Sumb](#) (Campion)
- [Cos](#) (Campion)
- [Next](#) (Infoarena)
- [Tort](#) (Infoarena)
- [Culori3](#) (Infoarena)



## **Legături**

Alți algoritmi care pot fi folosiți pentru înmulțirea a două numere mari:

- [Algoritmul Karatsuba](#)
- [Algoritmul Toom - Cook](#)



## **Bibliografie**

● Emanuela Cerchez, Marinel Șerban, “Programarea în limbajul C/C++ pentru liceu” volumul III, Editura Polirom, Iași, 2006

**Alexandru Cohal**  
[alexandru.cohal@yahoo.com](mailto:alexandru.cohal@yahoo.com)  
[alexandru.c04@gmail.com](mailto:alexandru.c04@gmail.com)  
Noiembrie 2013